Coverage Metrics for Model Checking

John Penix and Willem Visser[†]
Automated Software Engineering Group
[†]Research Institute for Advanced Computer Science (RIACS)
NASA Ames Research Center
{jpenix,wvisser}@ptolemy.arc.nasa.gov

Abstract

When using model checking to verify programs in practice, it is not usually possible to achieve complete coverage of the system. In this position paper we describe ongoing research within the Automated Software Engineering group at NASA Ames on the use of test coverage metrics to measure partial coverage and provide heuristic guidance for program model checking. We are specifically interested in applying and developing coverage metrics for concurrent programs that might be used to support certification of next generation avionics software.

1 Introduction

Model checking and testing are conceptually close neighbors, because they both operate over executable system models. In practice model checking is mostly used to analyze high-level requirements and design models of a system and testing is predominately used for the analysis of implementations. The most common link between these two techniques, suggested in the literature, has been to use model checking for test-case generation []. Recent advances in applying model checking to real programming languages [5, 8, 1, 10] has however opened up some interesting new ways in which testing and model checking can be used in tandem.

Model checking is often claimed to be 'better' than testing since all possible behaviors of a system are analyzed - the implication being that model checking might catch subtle errors that testing might miss. While this is true in theory, real systems tend to have very large state-spaces (more so than designs in general). To reduce the size of the state-spaces that must be searched, model checkers for programming languages typically use abstraction techniques. However effective abstraction often requires expert user input, and as such is not currently a solution that will find wide industrial appeal.

In case studies involving real systems, we have found that if an error exists it is often quite obvious (in hindsight) and one can make it appear by only considering a few subtle interactions rather than a multitude of complex ones (a.k.a the low hanging fruit principle). Hence, only looking at part of the state-space (or behaviors) can be very effective for finding errors when using a model checker to analyze a real program. Therefore, model checking programs is very similar to program testing: neither technique scales well to high levels of behavioral coverage for real systems and both can be effective at finding errors by examining a subset of program behaviors.

One notable difference between testing and model checking is that model checking is more suited to analysis of concurrent and reactive programs because it has full control over the scheduling of processes, threads and events, which is not typically the case in testing. Also, because a model checker stores the program states that it has visited, it can be more efficient in covering the behaviors of a program [2]. In addition, abstraction frameworks, such as abstract interpretation, can provide methods for constructing (conservative) over-approximations and m/c search optimizations can provide (conservative) under-approximations, both of which can be used to battle state space explosion while maintaining 'full coverage' or verification capabilities of the models.

In the following sections we present several ways in which program model checking can be improved by taking advantage of the close relationship to testing. In Section 2 we discuss structural coverage measures from testing that can measure partial coverage by model checking, and how these measures can be used to guide a model checker's execution. In Section 3 we show results obtained from extending the Java PathFinder model checker with the capability to calculate branch

coverage as well as use this coverage for guided search. Section 4 contain a short summary of the work presented and how we hope to proceed with this research.

2 Coverage for Model Checking

During testing it is common to use structural code coverage measures, such as decision (or branch) coverage, to obtain confidence that a program has been adequately tested. Coverage metrics include statement, decision (or branch) For example, the FAA requires software testing to achieve 100% modified condition/decision coverage (MC/DC) in order to certify level A criticality software for flight [7]. MC/DC coverage requires that all boolean conditions within a decision independently affects the outcome of the decision.

We are currently investigating whether similar coverage measures can be used when analyzing only a part of the state-space of a program during model checking. The simple answer is yes: the output of a model checker now becomes either that an error was found and a path that shows how to get to it, or if none is found it returns a coverage measure that testing engineers can interpret as is done now. The real question is figuring out why this could be useful. A model checker that can calculate traditional structural coverage could be useful in answering at least one interesting question that many people, including the FAA, has been struggling with for some time: how good is the MD/DC coverage measure at finding safety critical errors? [3]. But it would not be clear that a partial model checking result that includes 100% MC/DC coverage provides much of a guarantee of error-free operation. Note that achieving a certain structural coverage is not known to be useful in finding certain types of behavioral errors, such as timing/scheduling errors - i.e. exactly the ones model checking is good at finding.

Model checkers are particularly suited to finding errors in concurrent programs, but many traditional coverage criteria are essentially only meaningful for sequential programs. So, although it would be interesting to see how these measures work in the concurrent context, it may be more appropriate to investigate whether there are coverage measures more suitable to model checking. For example, the concurrent structural coverage measures from the testing literature, such as all-du-paths [9, 11], may be appropriate. A more interesting approach may be to develop suitable behavioral coverage measures. For example, "relevant path coverage" might be used to indicate coverage of the paths relevant to proving a property correct. Using behavior-based coverage metrics, it should be clear that program abstraction techniques, such as slicing and conservative abstraction, still provide full coverage even though some paths are not (completely) checked.

It is rather straight forward to argue that a model checker, during its normal operation, can calculate a coverage measure that can be used to evaluate how well the model checker did with respect to established testing measures. But, could these measures actually be used to improve model checking? For example, it is possible to guide the model checker to pick parts of the state-space to analyze based on structural coverage of the code that would generate those state-spaces. A simple example would be to consider only statement coverage: if the model checker can next analyze a program statement that has never executed, versus one that has, then it picks the new one.

3 Experiments with Java PathFinder

We have been experimenting with some of these ideas within the Java PathFinder (JPF) model checker[1] developed at NASA Ames. JPF is a model checker for Java programs that analyzes a program by executing the underlying bytecode instructions in a depth-first fashion. We modified the system such that it records the number of times a *true* and *false* branch in each branching instruction are taken. On the level of the bytecodes this amounts to doing decision (or branch) coverage - if a branch was taken more than 0 times then that decision was covered. The model checker displays its current coverage during execution. It quickly became apparent that this measure is quite useful to show progress within the model checker - the coverage would converge after some time passed, and after that would increase sporadically indicating some new code/behaviors are being executed.

But most of our experiments were done while analyzing multi-threaded Java programs, and we noticed that certain threads might achieve very low coverage although the overall coverage measure indicate high coverage for the complete program. We thus adapted the decision coverage to indicate coverage for each dynamically created thread in the program. Although this seemed to be the most obvious way to extend sequential coverage to concurrent coverage, we could not find any prior liter-

[1] Available from http://ase.arc.nasa.gov/jpf

ature and hence believe this is a novel approach. Although this thread-based coverage is very useful to see progress within the model checker, it is conservative when used as coverage measure. The problem is that certain threads cannot execute certain parts of the code, and hence low coverage is obtained even in the extreme case where the model checker can cover all behaviors of the program. We believe this problem can easily be overcome by doing static-analysis on the program before model checking to calculate more precisely what a thread's potential coverage should be.

To illustrate some of these concepts we will use the following Java program that contains a deadlock.

```
class Event{
  int count = 0;
  public synchronized void wait_for_event(){
    try{wait();}catch(InterruptedException e){};
  }
  public synchronized void signal_event(){
    count = count + 1;
    notifyAll();
} }

class FirstTask extends Thread{
  Event event1,event2;
  int count = 0;
  public FirstTask(Event e1, Event e2){
    this.event1 = e1; this.event2 = e2;
  }
  public void run(){
    count = event1.count;
    while(true){
      if (count == event1.count)
        event1.wait_for_event();
      count = event1.count;
      event2.signal_event();
} } }

class SecondTask extends Thread{
  Event event1,event2;
  int count = 0;
  public SecondTask(Event e1, Event e2){
    this.event1 = e1; this.event2 = e2;
  }
  public void run(){
    count = event2.count;
    while(true){
        event1.signal_event();
        if (count == event2.count) {
          event2.wait_for_event();
        }
        count = event2.count;
} } }

class Main {
  public static void main(String[] args){
    Event event1 = new Event();
    Event event2 = new Event();
    FirstTask  task1 = new FirstTask(event1,event2);
    SecondTask task2 = new SecondTask(event1,event2);
    task1.start(); task2.start();
} }
```

JPF cannot find the deadlock in this program, before exhausting memory, because it searched in a depth-first fashion and the *count* variable in the *Event* class is incremented indefinitely, hence creating a unique state every time. There are three threads in the program: *Main*, *FirstTask* and *SecondTask*. Both *FirstTask* and *SecondTask* have one decision point and hence two possible branches (4 branches in total). During execution the coverage soon converges to: 0 out of 4 for *Main* and 1 out of 4 for both the other two threads. But clearly *Main* does not even have any branching points, hence it in fact has full coverage, whereas the other two have 50% coverage each.

To achieve full coverage during testing of this example is hard since it is dependent on the scheduler. A model checker can however do it, by trying all interleavings. When the JPF feature to limit the length of depth-first paths is used, the error is discovered within seconds. In terms of coverage, the *FirstTask* stills has 50% coverage whereas *SecondTask* has 100% coverage when the error is found. This shows another unanticipated feature of the coverage measure: even when an error is found it might produce interesting results. Note in this case the one thread is still not fully covered, and closer inspection indicates that although the deadlock occurred due to a race condition that manifested itself in the *SecondTask*, the same thing could also have happened in the other thread. Admittedly, this is a trivial example, and this kind of information may be harder to interpret in real sized examples.

In order to experiment with the coverage-guided model checking idea we adapted the JPF scheduler to ignore threads in which the next statement is a branching statement which has already been taken n times before. For example, with $n = 10$, if the *true* branch is to be taken, but it has already been taken 10 times before, then ignore it and rather schedule another thread - the hope being that some other interleaving of statements will cause the *false* branch to be taken instead at some later point. This required a trivial change to the JPF scheduler.

If we now analyze the program from the previous section with this special branch-scheduling feature enabled (and with no depth limit) the error is found instantly (even with n as small as 1!). This is an extremely encouraging result, but the solution adopted is rather simplistic, and may ignore too much of the state-space. A

more general approach will be to order the statements to be executed next in the model checker according to whether or not they will improve coverage (rather than just ignoring statements that will not improve coverage). We plan to extend the JPF scheduler such that user-defined cost-functions can be added before model checking to rank transitions for execution. This will allow not just coverage measures to influence the ranking, but also other heuristics such as shortest path to a blocking statement (that might improve deadlock detection).

4 Conclusions

We have discussed the use of coverage measures for program model checking, where in practice analyzing all possible behaviors is not tractable. We have shown with a simple example that decision coverage for model checking can, not only show how well the model checker is doing, but also be used to guide a model checker.

The next step it to see how well these techniques work on industrial size problems. We are fortunate to have a Java version of the DEOS kernel from Honeywell that contains a subtle error. We, and others, have shown that this error can be discovered with model checking [6, 4]. We would now like to see how the coverage changes in cases where this error is found, and when it is not. This analysis of DEOS is ongoing and we hope to present preliminary results at the workshop.

REFERENCES

[1] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limeric, Ireland., June 2000. ACM Press.

[2] David Dill. Model checking java programs. In *ISSTA/FMSP Keynote Address*, 2000.

[3] Arnaud Dupuy and Nancy Leveson. An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. In *Proceedings of the Digital Avionics Systems Conference*, 2000.

[4] Matthew Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina Pasareanu, Robby, Willem Visser, and Hongjun Zheng. Tool-supported Program Abstraction for Finite-state Verification. In *Proceedings of the 23rd International Conference on Software Engineering (to appear)*, Toronto, Cananda., May 2001. ACM Press.

[5] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*. Springer-Verlag, September 2000.

[6] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, Limeric, Ireland., June 2000. ACM Press.

[7] RTCA Special Committee 167. Software considerations in airborne systems and equipment cer tification. Technical Report DO-178B, RTCA, Inc., dec 1992.

[8] SLAM. `http://www.research.microsoft.com/projects/slam/`.

[9] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.

[10] Willem Visser, Klaus Havelund, Guillaume Brat, and Seung-Joon Park. Model checking programs. In *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, September 2000.

[11] Cheer-Sun D. Yang, Arnie L. Souter, and Lori L. Pollock. All-du-path coverage for parallel programs. In *ISSTA: International Symposium on Software Testing and Analysis*, pages 153–162, 1998.